一种基于 LLVM 中间表示的数据依赖并行计算方法 *

朱 燕，衷璐洁

(首都师范大学 信息工程学院，北京 100048)

摘 要：底层虚拟机 LLVM 是一个广泛使用的编译框架，其中间表示 IR 中包含有丰富的程序分析信息，众多以 LLVM 为平台的相关工作均以 IR 为基础开展。数据依赖关系在错误检测、定位及程序调试等领域有着重要应用。基于 IR 的数据依赖关系计算多采用串行迭代方式，但在应对较大规模 IR 文件时可扩展性不够理想。对此，进行了数据依赖关系计算中指令读写的可并行性挖掘，结合图形处理器并行计算优势，提出一种基于 LLVM IR 的数据依赖关系并行计算方法 DRPC。该方法以 IR 为输入，采用 CPU-GPU 双端协同方式实现程序数据依赖关系的高效计算。实验结果表明，针对基准程序集 SPEC，DRPC 分别在直接及传递数据依赖关系计算上最高获得了 3.48x 和 4.91x 的加速比。

关键词：底层虚拟机；中间表示；数据依赖；图形处理器；可扩展性

中图分类号：TP301.4 doi: 10.19734/j.issn.1001-3695.2018.08.0536

Approach of parallel data dependence relation computation based on LLVM IR

Zhu Yan, Zhong Lujie

(*College of Information Engineering*, *Capital Normal University*, *Beijing* 100048, *China*)

Abstract: The low level virtual machine is a widely used compiler framework. Its intermediate representation contains abundant program analysis information, and many related work based on program analysis takes a basis of IR. Data dependence plays an important role in fault detection, fault location, and program debugging. IR-based data-dependence computing often adopts serial iterative approaches, such approaches usually suffer scalability issues when dealing with larger-scale IR contents. To address above issues, the parallelism mining of instructions reading and writing in data dependence calculation is carried out. Combined with the parallel computing advantages of graphics processing unit, a parallel computing method DRPC based on LLVM IR is proposed, which accepts IR as inputs, and achieves efficient program data dependence computation through a CPU-GPU cooperation way. The experimental results show that DRPC's highest speedup is 3.48x and 4.9x respectively in the direct and transitive data dependence computation with benchmark SPEC.

Key words: LLVM; intermediate representation; data dependence; graphics processing unit; scalability

0 引言

LLVM（low level virtual machine，底层虚拟机）是一个开源的编译框架，它提供一种基于 SSA（static single assignment）的中间表示（intermediate representation，IR），并以该表示作为后续优化及其他相关工作的基础[1]。IR 中包含有丰富的程序分析信息。数据依赖分析是一种重要的程序分析技术，在程序优化、维护及调试等众多方面有着广泛的应用[2~4]。如：以指令间的依赖关系为指导进行的指令调度及以程序实体间的依赖关系为依据实施的更精准错误定位等[5]。基于 LLVM IR 的数据依赖分析一般采用串行迭代的方式，这样的方式在运用于较大规模程序的数据依赖关系计算时可扩展性表现不够理想。

1 问题背景

1.1 数据依赖

数据依赖是一类典型的依赖分析方法，主要反映程序语句间需要遵守的读写依赖关系。根据语句间是否存在多个连续的数据依赖关系，可将数据依赖分为直接数据依赖及传递数据依赖两个类别。与数据依赖相关的定义如下：

定义 1 已知有 n 条有序执行的语句构成的语句序列 $S_1,S_2,\ldots,S_n$（$n\geq2$），其中，S_1 称为起点语句，S_n 称为终点语句，其余称为中间语句。相邻语句间的依赖关系 D_i（$1\leq i\leq n-1$）形成的依赖序列 $D=(D_1,D_2,\ldots,D_{n-1})$ 称为**依赖链**[6]。

将语句 S_i 的内存读操作变量集记做 read(S_i)，语句 S_i 的内存写操作变量集记做 write(S_i)。

直接数据依赖（依赖链长度为 2，即 n=2）的相关定义如下[7]：

定义 2 流依赖（flow dependence，FD）。若 write(S_i) ∩read(S_{i+1}) ≠Φ，i=1,2,…,n-1，则称 S_i 和 S_{i+1} 间存在流依赖，记做 S_i *FD* S_{i+1}。

定义 3 反依赖（anti dependence，AD）。若 read (S_i) ∩write(S_{i+1}) ≠Φ，i=1,2,…,n-1，则称 S_i 和 S_{i+1} 间存在反依赖，记作：S_i *AD* S_{i+1}。

定义 4 输出依赖（output dependence，OD）。若 write (S_i)∩write(S_{i+1}) ≠Φ，i=1,2,…,n-1，则称 S_i 和 S_{i+1} 间存在输出依赖，记做 S_i *OD* S_{i+1}。

特别地，当 n>2 时，若起点语句和终点语句间存在多个

收稿日期：2018-08-01；**修回日期：**2018-09-27 **基金项目：**国家自然科学基金资助项目（61872253，61402303）

作者简介：朱燕（1994-），女，广西柳州人，硕士学位，主要研究方向为程序分析（olt.123@163.com）；衷璐洁（1979-），女（通信作者），副教授，博士，主要研究方向为软件缺陷检测、先进编译技术等.

连续的、具有传递特征的数据依赖关系，则称这样的关系为传递依赖。

定义 5　传递依赖（transitive dependence，TD）。对于语句序列 S_i , S_{i+1} , S_{i+2} ,…,S_j，若语句 S_i 和 S_j 间满足如下条件：

$write(S_m)\cap read(S_{m+1}) \neq \Phi \ (i \le m \le j-1)$，

则称 S_i 和 S_j 间存在传递依赖关系，记做 $S_i\ TD\ S_j$。值得注意的是，由于存在传递依赖关系的语句序列中相邻语句间均存在流依赖关系，因此该序列也将形成一条**流依赖链**。

以图 1(a)所示程序片段为例，由于 $write(S_2) = \{a\}$，$read(S_3)=\{a\}$，$write(S_3)=\{b\}$，$read(S_5)=\{b\}$，$write(S_5)=\{d\}$，有 $Write(S_2)\cap Read(S_3)\neq\Phi$ 且 $write(S_3)\cap read(S_5) \neq\Phi$，因此，在 S_2 与 S_5 间存在传递依赖关系。

```
int func()
{
    S1:  int a,b,c;
    S2:  a = 1;
    S3:  b = a + 1;
    S4:  c = a + 3;
    S5:  int d = b + 5;
    S6:  if(d > 10){
    S7:      return c; }
    S8:  else{
    S9:      return a; }
}
```

(a)

```
; Function Attrs: nounwind uwtable
define i32 @func() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  %d = alloca i32, align 4
  store i32 1, i32* %a, align 4
  %0 = load i32, i32* %a, align 4
  %add = add nsw i32 %0, 1
  store i32 %add, i32* %b, align 4
  %1 = load i32, i32* %a, align 4
  %add1 = add nsw i32 %1, 3
  store i32 %add1, i32* %c, align 4
  %2 = load i32, i32* %b, align 4
  %add2 = add nsw i32 %2, 5
  store i32 %add2, i32* %d, align 4
  %3 = load i32, i32* %d, align 4
  %cmp = icmp sgt i32 %3, 10
  br i1 %cmp, label %if.then, label %if.else…}
```

FD　读内存　写内存

①先写后读%a　②先写后读%b　③先写后读%d

(b)

图 1　内存读写型数据依赖关系示例

Fig. 1　Example of memory read and write data dependence computation based on LLVM IR

1.2　LLVM IR

图 2 给出了 LLVM 编译框架的构成示意，主要包含三个部分：前端、优化器及后端。LLVM 自定义的中间表示 IR 是该框架的重要组成部分。IR 不仅是 LLVM 优化及后端相关工作的基础，也是基于 LLVM 程序分析的各种应用的主要输入。IR 一般有三种形式：可读汇编形式、二进制码形式及内存中间表示形式。

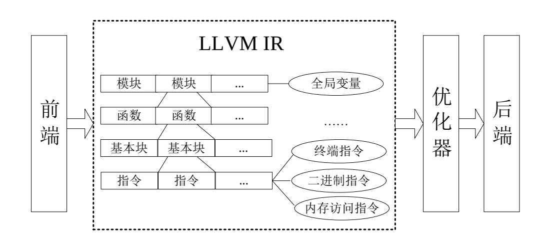


图 2　LLVM 编译框架

Fig. 2　LLVM compilation framework

LLVM IR 内容文本通常以指令集形式提供包含函数声明、基本块、变量声明及函数调用等在内的各项程序分析信息。LLVM 典型的指令集类别包括内存访问指令、终端指令、二进制指令、按位二进制指令以及其他指令等。在图 1(b)中给出了图 1(a)所示代码片段的 IR 文本内容示例。

1.3　基于 LLVM IR 的数据依赖计算

在 LLVM IR 中，与数据依赖计算相关的内存读写指令包括 alloca、store 和 load 等。其中 alloca 指令表示变量声明，store 指令表示变量的内存写操作，load 指令表示变量的内存取操作，并且三种指令中的相关变量会以符号"%"进行标志。通过对相关读写指令的特征追溯，可以提取出数据依赖关系计算所需的程序分析信息。例如可通过追溯 IR 中的 *store* 和 *load* 指令并实施分析，计算获得数据依赖关系计算所需的 *Write* 集和 *Read* 集，进而完成数据依赖关系的相关计算。

以图 1(a)所示程序片段为例，在 IR 中识别变量 *a*、*b*、*d* 相关的内存读写指令，用 S_{store1} 表示图 1(b)中标志①相关的 *store* 指令，S_{load1} 表示图 1(b)中标志①相关的 *load* 指令。可知 $Read(S_{store1})=\{1\}$，$Write(S_{store1})=\{\%a\}$，$Read(S_{load1})=\{\%a\}$，$Write(S_{load1})=\{\%0\}$，由于 $Write(S_{store1})\cap Read(S_{load1}) \neq\Phi$，进而可计算出 S_{load1} 和 S_{store1} 间的流依赖关系。其特征追溯路线参见图 1(b)中的带箭头弧形虚线。

1.4　基于 IR 的数据依赖关系计算并行性分析

直接数据依赖关系计算中的并行性. 直接数据依赖关系的计算需要考虑起点指令和终点指令间的相关读写交集。一方面，可利用不同起点指令与终点指令间的非连续读写交集特征进行计算任务划分，另一方面，也可充分利用相关指令信息的读取及数据依赖关系更新的访存特性进一步获得数据并行性。

传递数据依赖关系计算中的并行性. 传递数据依赖关系的计算虽然具有动态性，但仍然具备较好的数据并行特征，例如，可调度多个线程同时计算不同起点指令、终点指令与同一中间指令之间的读写交集。

本文提出一种基于 LLVM IR 的、以 GPU 为环境的数据依赖关系并行计算方法，通过 CPU-GPU 双端联合的方式实现数据依赖关系的高效计算，具体包括：

a)一种面向数据依赖关系计算的 CPU-GPU 数据映射方法，在有效隐藏数据映射延迟的同时，进一步保障可扩展性目标的实现。

b)基于多流协同的直接数据依赖关系计算方法及 GPU 访存敏感的双端协同传递数据依赖关系计算方法。在数据依赖关系计算的过程中，针对不同迭代间的耦合关系规避问题，在保障计算准确性的同时实现 CPU 端与 GPU 端之间的计算同步。

c)结合 GPU 存储介质特点及数据依赖关系计算任务访存特性考虑的数据分布策略，进一步提升访存效率。

2　基于 LLVM IR 的数据依赖并行计算框架

本文主要围绕三个影响数据依赖并行计算的可扩展性问题：非规则存取带来的带宽资源浪费问题；CPU-GPU 双端的高效协同问题；线程不同步破坏传递数据依赖计算的动态性问题，提出基于 LLVM IR 的数据依赖关系并行计算框架(data Dependence Relation Parallel Computation　framework，DRPC)。如图 3 所示，主要由 IR 特征指令匹配及信息提取(Pre_Match_Extract)、数据适配存储映射（Info_Storage）、直接依赖关系计算(Direct_DepCal)、传递依赖关系计算(Trans_DepCal)和并行优化(Parallel_Opt)及数据依赖关系应用输出(DepOutput_forApp)几个部分组成。

其中，Pre_Match_Extract 主要负责基于特征制导的关键指令匹配及相关程序分析信息的提取；Info_Storage 主要负责根据直接依赖关系及传递依赖关系计算需要，实现 CPU 端及 GPU 端的数据信息映射；Pre_Match_Extract 和 Info_Storage 是 DRPC 的基础性组成。在适应性数据映射完成后，由 Direct_DepCal 和 Trans_DepCal 共同负责完成各类数据依赖关系的并行计算。其中，Direct_DepCal 主要负责直接数据依赖关系的并行计算，Trans_DepCal 负责在直接数据依赖关系计算完成的基础上继续完成传递数据依赖关系的并行计算。

在数据依赖关系的并行计算过程中，Parallel_Opt 会负责实施进一步的性能优化。最后由 DepOutput_forApp 根据后续应用需求对数据依赖关系进行组织和输出。

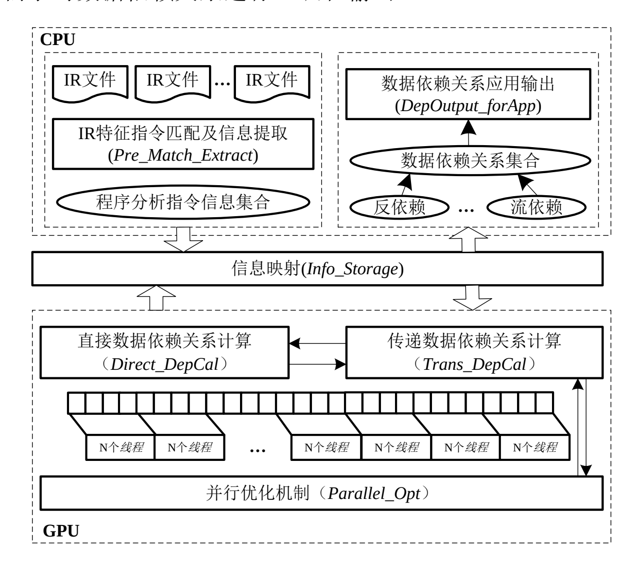


图 3　基于 llvm ir 的数据依赖关系并行计算框架

Fig. 3　DRPC framework

同时，本文使用 CUDA 作为 *DRPC* 的 GPU 并行计算环境。CUDA 的线程模型被划分为三个层次，分别为线程格（grid）、线程块（block）以及线程（thread）。每个线程格内包含一定数量的线程块，每个线程块内包含一定数量的线程，并且每个线程、线程块和线程格都拥有编号。相同线程块中的线程可访问同一个共享内存区域，所有线程都可以访问全局内存。在进行数据依赖关系计算时，DRPC 将待计算数据按线程数量划分，并利用线程编号进行数据索引，让多个线程同时进行数据依赖关系计算。

3　DRPC 具体实现

3.1　存储设计

1）指令集存储设计

三个向量 var_arr、inst_read_arr 和 inst_write_arr 分别负责指令 alloca、store 及 load 的 Write 集和 Read 集内容存储。其中，var_arr 负责存储 IR 中的变量信息，其向量下标对应变量编号。inst_read_arr 和 inst_write_arr 负责存储内存读写操作变量的信息，其下标对应指令索引。同一指令中的读写变量在向量 inst_read_arr 和 inst_write_arr 中具有相同下标，特别地，指令的内存常量读取设置特殊值-1。图 4 给出了图 1 示例的指令信息存储示意。以图 1 的第一个 store 指令为例，变量%a 在 var_arr 中的下标为 0，则该下标也是%a 的变量编号。那么在向量 inst_write_arr 下标为 0 的位置中，存入%a 的编号 0。在向量 inst_read_arr 下标为 0 的位置中，则存入表示常量 1 的特殊值-1。

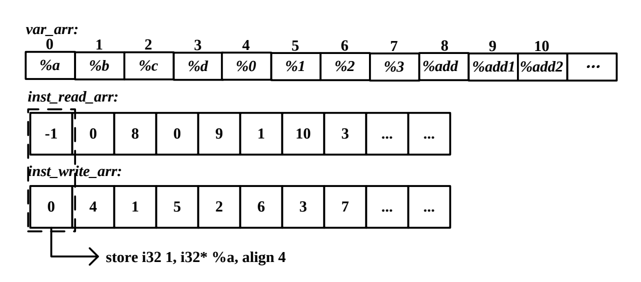


图 4　特征指令集存储设计

Fig. 4　Design of characteristic instructions set storage

2）数据依赖关系存储

数据依赖信息以矩阵 *Matrix* 方式存储，*Matrix* 的定义如式（1)所示。

$$Matrix[i,j]=\{ND,FD,AD,OD,TD\} \quad (i,j=0,1,2,\ldots,n\text{-}1) \tag{1}$$

其中：*i*，*j* 表示指令 $Inst_i$ 和 $Inst_j$ 的指令索引。取值 *ND*、*FD*、*AD*、*OD* 和 *TD* 分别表示 $Inst_i$ 和 $Inst_j$ 间不存在数据依赖关系、存在流依赖关系、存在反依赖关系、存在输出依赖关系及存在传递依赖关系。

3.2　特征制导的 IR 指令预处理

DRPC 中的 *Pre_Match_Extract* 主要负责 IR 指令的分析预处理工作。首先，根据特征关键词 *alloca*、*store*、*load*，从 IR 文本中筛选出待分析指令。之后根据指令的组织模式提取数据依赖计算所需的信息。同时，本文主要关注 IR 级别的指令间的数据依赖关系，对于循环结构，本文直接实施指令信息提取，并分析两两指令间的数据依赖关系。

表 1　待分析指令的简要组织模式及描述

Table1 Brief organization mode and description of the instructions to be analyzed

指令组织模式	描述
store *OPType* OP_1, *OPType* OP_2	读取 OP_1 的值，并赋给 OP_2
OP_2= load *OPType*, *OPType* OP_1	OP_1 从内存中加载数据，并将该值赋给 OP_2
OP_1= alloca *OPType*	声明变量 OP_1，并在内存中为 OP_1 开辟空间

表 1 归纳了三种待分析指令的组织模式，并给出了相应描述，其中 OP_1 与 OP_2 代表指令操作对象，*OPType* 代表变量的数据类型。

Pre_Match_Extract 主要工作过程的算法描述如算法 1 所示，其中 *Match*(*x*,*y*)表示对传入的原始 IR 指令 *x* 进行关键指令匹配，并将匹配所得的指令类型返回给 *y*。算法 1 将逐行处理 IR 原始信息文本，通过调用 *Match*(*Inst*, *Inst_Type*) 完成 IR 中关键指令的匹配（行 1-2）；若匹配为 *alloca* 指令，则将所提取的指令关键信息存入 *var_arr*（行 3-5）；若匹配为 *store* 或 *load* 指令，则存入 *inst_read_arr* 或 *inst_write_arr*（行 6-9）。

算法 1　Pre_Match_Extract

输入：IR 信息

输出：var_arr、inst_read_arr 以及 inst_write_arr

```
1. for each Inst in IR
2. Match( Inst, Inst_Type );
3. if Inst_Type == alloca
4.     var_arr←alloca 指令信息;
5. end if
6. if Inst_Type == ( store || load )
7.     inst_read_arr←读操作变量信息;
8 inst_write_arr←写操作变量信息;
9. end if
10. end for
```

3.3　基于多流协同的基础数据依赖关系计算

在数据依赖关系计算部分，*DRPC* 首先由 *Direct_DepCal* 完成直接的数据依赖关系计算，主要包括流依赖、反依赖和输出依赖等依赖关系的计算。*Direct_DepCal* 采用多流协同的方式进行依赖关系的计算。图 5 给出了基于多流协同的直接数据依赖关系计算的工作机制示意。

Stream 0: | Info_Storage | GPU_Direct_Kernel | Data_Return | ……
Stream 1: | | Info_Storage | GPU_Direct_Kernel | Data_Return | ……

图 5 多流协同计算的工作机制

Fig. 5 Multi-stream collaborative computing mechanism

两个流 Stream0 及 Stream1 以流水线方式分别执行数据复制、内核函数执行及计算结果的返回。例如，在 Stream0 完成数据映射工作后，将开始执行 GPU_Direct_Kernel 进行直接依赖关系的并行计算，与此同时 Stream1 调用 Info_Storage 进行数据映射。当 Stream0 结束 GPU_Direct_Kernel 执行后，会将直接数据依赖信息回传至 CPU 端，同时 Stream1 启动 GPU_Direct_Kernel。其中，GPU_Direct_Kernel 通过计算 inst_read_arr 和 inst_write_arr 的交集来获得直接数据依赖关系，具体计算方法如式(2)所示。在式 2)中，$Inst_s$ 和 $Inst_e$ 分别代表起点指令和终点指令，s 和 e 分别对应这两条指令的索引。

$$Matrix[s,e]=\begin{cases} FD,\ 若inst_write_arr[s]==inst_read_arr[e] \\ AD,\ 若inst_read_arr[s]==inst_write_arr[e] \\ OD,\ 若inst_write_arr[s]==inst_write_arr[e] \end{cases} \qquad (2)$$

算法 2 给出了 CPU_Direct_DepCal 的算法描述，此算法主要负责 CPU 端的相关工作。其中，Info_Storage(x)表示由流 x 完成数据映射工作；GPU_Direct_Kernel(x)表示由流 x 执行 GPU 内核函数；Data_Return(x,y,z) 表示由流 x 将 y 回传至 z。对于程序中的每一个函数 f，CPU_Direct_DepCal 首先调用 Info_Storage(Stream0)，由 Stream0 进行所需的数据映射工作，为 inst_read_arr 及 inst_write_arr 在 GPU 端开辟存储空间 d_Read_Arr 和 d_Write_Arr，以及为直接数据依赖关系集 Dep_{direct} 开辟 GPU 端存储空间 d_Matrix（行 1-2）；随后调用 GPU_Direct_Kernel(Stream0)进行直接数据依赖关系计算（行 3）；并在计算完成后调用 Data_Return(Stream0,d_matrix,Dep_{direct})，由 Stream0 将 d_Matrix 回传至集合 Dep_{direct}，并同时启动 Info_Storage(Stream1)进行所需的数据映射工作（行 4）；之后调用 GPU_Direct_Kernel(Stream1) 及 Data_Return(Stream1,d_matrix, Dep_{direct})，由 Stream1 调用核函数进行下一次计算及数据回传（行 5-6）。

算法 2 CPU_Direct_DepCal

输入： inst_read_arr、inst_write_arr

输出： 直接数据依赖关系集 Dep_{direct}

1. for each f do
2. Info_Storage(Stream0);
3. GPU_Direct_Kernel(Stream0);
4. Data_Return(Stream0,d_Matrix,Dep_{direct})&& Info_Storage(Stream1);
5. GPU_Direct_Kernel(Stream1);
6. Data_Return(Stream1,d_Matrix,Dep_{direct});
7. end for

算法 3 给出了 GPU_Direct_Kernel 的算法描述，其中 Share_W 和 Share_R 是为了减少访存延迟而设置的共享数组，用于存储 d_Read_Arr、d_Write_Arr 中的数据，函数 Init(Array, x, y)表示利用索引信息 x 和 y 初始化数组 Array。算法 3 首先通过线程号、线程块号计算起点指令 $Inst_s$ 和终点指令 $Inst_e$ 的指令索引 s 和 e（行 1），然后调用 Init(Share_W, s, e)和 Init(Share_R, s, e)将 d_Write_Arr、d_Read_Arr 中的数据映射至 Share_W 以及 Share_R（行 2-3），随后根据式 2)计算直接数据依赖关系，最后存储至 d_Matrix（行 4-12）。

算法 3 GPU_Direct_Kernel

输入： d_Read_Arr、d_Write_Arr 以及 d_matrix

输出： 包含直接数据依赖信息的集合 d_matrix

1. 根据线程号及线程块号获得指令索引 s、e
2. Init(Share_W, s, e);
3. Init(Share_R, s, e);
4. if Share_W[s] == Share_R[e]
5. d_Matrix[s, e]←FD;
6. end if
7. if Share_R[s] == Share_W[e]
8. d_Matrix[s, e]←AD;
9. end if
10. if Share_W[s] == Share_W[e]
11. d_Matrix[s, e]←OD;
12. end if

3.4 GPU 访存敏感的双端协同传递数据依赖关系计算

在 *Direct_DepCal* 计算结果的基础上，*DRPC* 将由 *Trans_DepCal* 完成传递数据依赖关系的计算。在该阶段，需要先进行流依赖链的寻找。对于有序执行的指令序列 $Inst_i, Inst_{i+1}, \ldots, Inst_j (j>i)$，根据指令索引 *i*，*j* 判断 $Inst_i$ 与 $Inst_j$ 之间是否存在流依赖链的方法如式 3)所示。

$$\begin{aligned} &Matrix[i,\ i+1]==FD \\ &\wedge Matrix[i+1,i+2]==FD \\ &\wedge \ldots \wedge Matrix[j-1,j]==FD \end{aligned} \qquad (3)$$

流依赖链的构建是一个动态的过程，其长度会随中间指令的不断加入而增长。算法 4 和 5 分别给出了 CPU_Trans_DepCal 及 GPU_Trans_Kernel 的算法描述。其中，函数 Map_midNumber(*x*)负责映射中间指令 $Inst_x$ 的指令索引 *x*；*GPU_Trans_Kernel*(*k*)表示向核函数 GPU_Trans_Kernel 中传入参数 *k*。在算法 4 中，首先为 Dep_{direct} 分配 GPU 端存储空间 *d_Trans_Mat*（行 2）；依次处理中间指令集 $S_{midinst}$ 中的各条指令 $Inst_k$，获得 GPU 端映射中间指令索引 *k*；然后调用 *GPU_Trans_Kernel*(*k*)，构建并分析含有中间指令的依赖链（行 3-6）。计算结束后将 GPU 端的 *d_Trans_Mat* 回传至 Dep_{trans}（行 7）。

算法 4 CPU_Trans_DepCal

输入： 直接数据依赖信息集 Dep_{direct}

输出： 传递数据依赖信息集 Dep_{trans}

1. for each f do
2. d_Trans_Mat←Dep_{direct};
3.   for each $Inst_k$ in $S_{midinst}$ do
4.   k←Map_midNumber($Inst_k$);
5. GPU_Trans_Kernel(k);
6.   end for
7. Dep_{trans}←d_Trans_Mat;
8. end for

在算法 5 中，首先获取线程索引 threadId 和线程块索引 blockId（行 1）；之后利用上述信息调用 Init(Share_k, threadId, blockId)初始化共享数组 Share_*k*（行 2）；并通过寻找以 *blockId* 为起点指令索引，*threadId* 为终点指令索引的流依赖链来计算传递数据依赖关系（行 3-5）。

算法 5 GPU_Trans_Kernel

输入：包含直接数据依赖关系的 d_Trans_Mat 及 k

输出：包含传递数据依赖关系的 d_Trans_Mat

1. 获取线程索引 threadId 和线程块索引 blockId
2. Init(Share_k, threadId, blockId);
3. if d_Trans_Mat[blockId, k] == Share_k[k, threadId] then

4. d_Trans_Mat[blockId, threadId]←TD;
5. end if

3.5　并行优化机制

在 Trans_DepCal 阶段，Parallel_Opt 会就异构计算同步及数据分布等问题进行有针对性的优化。1）在传递数据依赖关系计算时，会实施中间指令添加工作，即在起点指令和终点指令之间添加中间指令以完成依赖链的构建与分析。每对一个新的依赖链进行分析，都需要依靠其余依赖链的分析结果，所以新依赖链的构建必须在其余依赖链完成计算的基础上进行。为了保证新依赖链分析的准确性，双端协同会在 GPU 端的本次分析结束之后，再从 CPU 端传入构建新依赖链所需的数据，从而使线程同步，避免遗漏其余依赖链的计算结果。2）考虑到访存效率，数据依赖关系以 matrix 形式存储在全局内存中，并在数据划分时利用 GPU 全局内存的合并访问机制[8]，对 matrix 按条带状划分，并建立中间指令索引 k 与起点指令索引之间的索引映射。在这样的方式下，一个线程束访问的全局内存地址范围如式 4)所示：

$$range = \{k * N + j \mid j \in \{0 \ldots 31\}\} \tag{4}$$

其中，N 代表 *Matrix* 宽度，j 代表线程号，同时映射为终点指令索引。内存地址连续的 *range* 可充分利用全局内存的合并访问优势，减少对全局内存的访问次数。此外，由于中间指令与其他指令形成的数据依赖关系被访问的频率较高，故将此部分数据存储在共享内存中，以提高数据访问的局部性，进而提升访存效率。

3.6　数据依赖关系应用输出

数据依赖关系计算完成之后，*DRPC* 可根据后续应用的不同需求进行数据依赖关系的组织和输出。图 6 给出一种图 1(b)的数据依赖图输出示例。

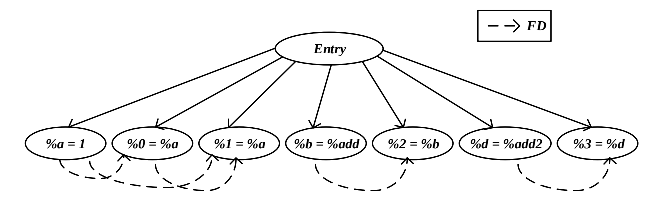


图 6　图 1(b)的数据依赖图

Fig. 6　Data dependence graph of Figure 1(b)

其中椭圆节点代表指令，并且指令以简单等式的形式组织，等式左边代表指令中进行写操作的变量，等式右边代表指令中进行读操作的变量；*Entry* 为入口指令节点；带箭头弧形虚线代表指令间的流依赖关系。

4　实验结果及分析

4.1　实验环境

本文实验环境配置为 Intel Core i7 CPU+NVIDIA GeForce GTX 1050Ti。实验基准程序为 SPEC 2000，所有实验均以 LLVM IR 文件为输入。

4.2　实验结果

实验选取了 SPEC 2000 中的 254.gap、183.equake、186.crafty 以及 176.gcc 程序作为实验用例。表 2 给出了上述实验用例中 store 指令和 load 指令数量的统计结果，二者在数据依赖相关指令总数中占比均超过 81%。同时，结合 GPU 端全局内存的合并访存机制以及同线程块内的数据共享，通过线程与指令的一一映射，实现了一个线程束同时访问连续 32 条指令的支持，进一步提高了全局内存的访存效率。当指令数量较少时，线程数量较少，导致计算任务的并行度较低，此时线程对全局内存的访问无法友好地隐藏线程的数据依赖计算操作，合并访存效果不佳。当指令数量较大时，活跃线程较多，数据访问能够较好地隐藏访存延迟，线程能够获取的共享数据资源增多，任务并行度和资源并行度都表现较好。

4.2.1 时间开销

表 2 给出了 *DRPC* 的数据依赖关系计算时间数据，其中 *SDIR* 和 *STRA* 分别表示未并行化（串行）的直接数据依赖和传递数据依赖关系计算的时间数据；*PDIR* 和 *PTRA* 分别表示本文提出的直接数据依赖关系并行算法和传递数据依赖关系计算并行算法的运行时间数据，并且 *PDIR* 和 *PTRA* 包含 CPU 端向 GPU 端的数据传输时间；图 7 为直接数据依赖关系计算串并行时间比较。图 8 为传递数据依赖关系串并行时间比较。由于 *254.gap* 及 *183.equake* 的数据与 *186.crafty* 及 *176.gcc* 的数据存在数量级差异，故分别以(a)和(b)两部分给出。此外，图 9 还给出了直接数据依赖关系和传递数据依赖关系计算的加速比数据。在图 9 中，*DIRS* 和 *TRAS* 分别对应直接数据依赖关系计算和传递数据依赖关系计算的加速比。

表 2　数据依赖计算时间数据

Table 2 Data dependence computation experiment time data

实验用例	IR 大小 (KB)	IR 行数 (LOC)	*store* 及 *load*（LOC）	*SDIR* (ms)	*PDIR* (ms)	流依赖 (LOC)	反依赖 (LOC)	输出依赖 (LOC)	*STRA* (ms)	*PTRA* (ms)	传递依赖 (LOC)
254.gap	201	4662	1600	7.05	2.43	1550	1255	584	1148.96	634	180
183.equake	347	6934	2688	21.97	9.6	3636	2951	944	19069.93	4197	249
186.crafty	547	10676	2816	33.26	12.37	7287	6038	5230	102517	22228	353
176.gcc-1	1,913	34663	10892	163.35	53.72	6150	3491	2697	186540.55	37964	1247
176.gcc-2	3,193	54772	16010	294.33	84.66	20732	13823	6324	15565.42	3481	8521

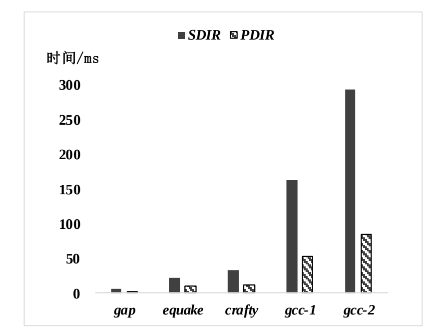
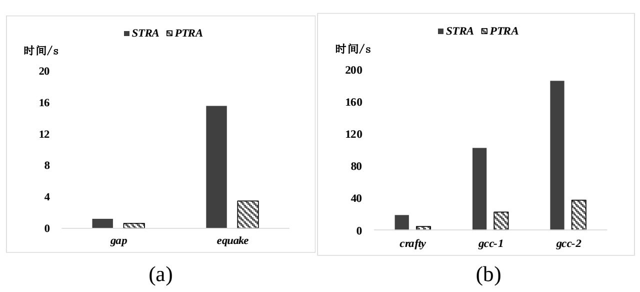


图 7　直接数据依赖关系计算串并行时间比较

Fig. 7　Serial time and parallel time comparison of direct data dependence computation



(a)　　(b)

图 8　传递数据依赖关系计算串并行时间比较

Fig. 8　Serial time and parallel time comparison of transitive data dependence computation

分析图 7 可知，对所有实验程序，*PDIR* 的在时间性能上均优于 *SDIR*。当 IR 信息文本规模较大时，*PDIR* 相较于 *SDIR* 表现出了更为明显的时间优势。直接数据依赖关系计算的加速比由 2.29x 升至 3.48x。这主要是由于活跃线程的数量增多，多流协同计算的策略有效地隐藏复制数据带来的延迟

所带来的性能提升。

在图 8 中，当实验数据规模较小时，图 8(a)中 *STRA* 与 *PTRA* 时间差值较小，加速比仅为 1.8x，这主要是由于数据量较小，GPU 端活跃线程数量较少，传递依赖关系计算未能很好地隐藏访存延迟。而当实验数据规模增大，在图 8(b)中，由于合并访问机制及共享内存的合理使用，访存效率提高，同时活跃线程的增多较好地隐藏了访存延迟，令传递闭包并行计算的加速比提升至了 4.91x。

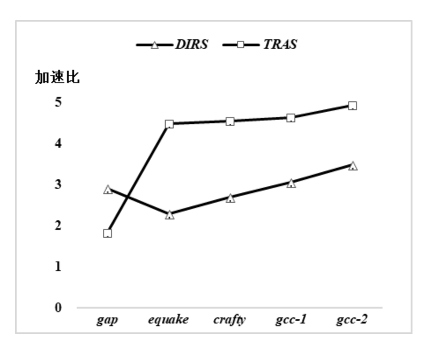


图 9　加速比数据

Fig. 9　Speedup of data dependence calculation

4.2.2 空间开销

如表 3 所示，*SDDC* 与 *PDDC* 分别代表了未并行化的数据依赖计算和本文提出的数据依赖并行计算的空间数据。其中，*SDDC* 只涉及到 CPU 的内存使用，而 *PDDC* 的 CPU 端空间数据包含了存储在页锁定内存的数据，该部分的数据用于直接数据依赖并行计算中多流协同机制的数据复制，所以 *PDDC* 在 CPU 端的空间开销上相较于 *SDDC* 要多出约 0.8%。除此之外，*PDDC* 还包括 GPU 端并行计算所需数据的内存开销，表 3 给出了 GPU 端的拷贝开销。由表可知 GPU 端的拷贝开销最高占 *PDDC* 的 4.9%，这是由于多流协同机制使得 GPU 端只需要分配一个函数中的指令的所需空间，所以总体而言，*SDDC* 的空间开销接近 *PDDC*。

表 3　数据依赖计算空间数据

Table 3 Data dependence computation experiment space data

实验用例	*SDDC*(M)	*PDDC*(M)	
		CPU	GPU
254.gap	10.08	10.16	0.25
183.equake	56.43	56.60	1.56
186.crafty	52.94	53.07	2.64
176.gcc-1	274.01	274.64	2.85
176.gcc-2	450.94	451.88	3.52

5　相关工作

数据依赖分析方法. Johnson 等人[9]提出了一种基于 LLVM 的协作依赖分析框架，该框架对于不同的依赖场景使用特定的依赖分析算法，同时获得了准确性与依赖分析方法的可组合性。高念书等人[10]提出一种实用的数据依赖分析方法，该方法根据数组下标的不同类型采用不同的依赖分析方法，解决循环正规化的一些限制问题。姜淑娟等人[11]提出了一种基于异常传播分析的依赖分析方法，通过建立一种能够描述异常处理结构的系统依赖图来适应含有异常处理的 C++ 程序的分析，解决了忽略异常处理而造成的分析不准确问题。Nikolaos 等人[12]提出了一种递归并行任务之间的动态数据依赖的系统 PARTEE，该系统通过分析内存踪迹确定任务间的依赖关系，实验表明 PARTEE 比 Nanos++能解决更多细粒度问题，且在任务依赖不规则的情况下，PARTEE 比 Cilk 高出 103％。Litvak 等人[13]提出了一种域敏感的传递依赖关系分析算法，解决了大型结构体变量之间传递依赖关系的分析问题，该算法在内存消耗上降低了 31%。

并行化的数据依赖分析. Minjang Kim 等人[14]提出了一种可扩展的动态数据依赖分析框架 SD^3，该框架利用流水线和数据并行的混合并行模型进行数据依赖分析，减少了时间开销，实验表明在 8 核和 32 核处理器上分别获得了 4.1x 和 9.7x 的加速比。Yu 等人[15]提出一种将分析任务划分为独立片的方法，该方法通过并行分析不同的片生成子图，最终由编译器自动将其组合成完整的数据依赖图。实验表明，对于多个知名的基准程序，该并行方案将分析时间缩短了几个数量级。Li 等人[16]提出了一种无锁的并行数据依赖分析方法，该方法为每个内存地址分配一个处理存储器访问的工作线程，同时线程并行执行分析算法，该方法相较于其未并行化的数据依赖分析方法获得了 2.1x-2.4x 的加速比。

6　结束语

针对基于 IR 的数据依赖计算的可扩展性问题，本文提出了一种基于 LLVM 中间表示的数据依赖并行计算框架 DRPC。该框架对传统迭代方式的并行性进行挖掘，并解决了基于 IR 的数据依赖计算在 GPU 端中的数据适配性问题与同步性问题，根据数据访问频率，合理分配数据，同时提高任务并行性，实验证明该方法能有效提高基于 IR 的数据依赖计算的可扩展性。

参考文献：

[1] Lattner C. The architecture of open source applications: LLVM [EB/OL]. (2012) [2018-8-1]. http://www. aosabook. org/en/llvm. html.

[2] 王涛，韩兰胜，付才，等. 软件漏洞静态检测模型及检测框架 [J]. 计算机科学, 2016, 43(5): 80-86. (Wang Tao, Han Lansheng, Fu Cai, *et al.* Static detection model and framework for software vulnerability [J]. Computer Science, 2016, 43 (5): 80-86.)

[3] Bates S, Horwitz S. Incremental program testing using program dependence graphs [C]//Proc of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages. New York: ACM Press, 1993: 384-396.

[4] Binkley D. Using semantic differencing to reduce the cost of regression testing [C]//Proc of Conference on Software Maintenance. Piscataway, NJ: IEEE Press, 1992: 41-50.

[5] 王克朝，王甜甜，苏小红，等. 软件错误自动定位关键科学问题及研究进展 [J].计算机学报，2015，38(11): 2262-2278. (Wang Kechao, Wang Tiantian, Su Xiaohong, *et al.* Key scientific issue and state-art of automatic software fault localization [J]. Chinese Journal of Computers, 2015, 38(11) , 2262-2278.)

[6] 缪力，张大方. 扩展有限状态机 EFSM 的后向切片[J]. 软件学报，2004, 15(suppl): 169-178. (Miao Li, Zhang Dafang. Computing backward slice of EFSMs [J]. Journal of Software, 2004, 15(suppl): 169-178.)

[7] 吴忧. 基于 LLVM 的 C 程序的动态数据依赖分析工具的设计与实现 [D]. 长春：吉林大学, 2016. (Wu You. The design and implementation of dynamic data dependence analysis tool for C program based on LLVM [D]. Changchun: Jilin University, 2016.)

[8] Nicholas W. CUDA 专家手册: GPU 编程权威指南 [M]. 苏统华，马培军，译. 北京：机械工业出版社, 2014: 97-98. (Nicholas W. CUDA handbook: a comprehensive guide to GPU programming [M]. Beijing: China Machine Press, 2014: 97-98.)

[9] Johnson N P, Fix J, Beard S R, *et al*. A collaborative dependence analysis framework [C]//Proc of IEEE/ACM International Symposium on Code Generation and Optimization. Piscataway, NJ: IEEE Press, 2017: 148-159.

[10] 高念书，张兆庆，乔如良. 实用数据依赖分析方法 [J]. 计算机学报, 1995, 18(4): 258-265. (Gao Nianshu, Zhang Zhaoqing, Qiao Ruliang. Practical data dependence analysis [J]. Chinese Journal of Computers, 1995, 18(4): 258-265.)

[11] 姜淑娟，徐宝文，史亮，等. 一种基于异常传播分析的依赖性分析方法 [J]. 软件学报, 2007, 18(4): 832-841. (Jiang Shujuan, Xu Baowen, Shi Liang, *et al*. An approach to analyzing dependence based on exception propagation analysis [J]. Journal of Software, 2007, 18(4): 832-841.)

[12] Papakonstantinou N, Zakkak F S, Pratikakis P. Hierarchical parallel dynamic dependence analysis for recursively task-parallel programs [C]//Proc of Parallel and Distributed Processing Symposium. Piscataway, NJ: IEEE Press, 2016: 933-942.

[13] Litvak S, Dor N, Bodik R, *et al*. Field-sensitive program dependence analysis [C]//Proc of ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM Press, 2010: 287-296.

[14] Kim M, Kim H, Luk C K. SD3: A Scalable Approach to Dynamic Data-Dependence Profiling [C]//Proc of IEEE/ACM International Symposium on Microarchitecture. Piscataway, NJ: IEEE Press, 2010: 535-546.

[15] Yu Hongtao, Li Zhiyuan. Multi-slicing: a compiler-supported parallel approach to data dependence profiling [C]//Proc of International Symposium on Software Testing and Analysis. New York: ACM Press, 2012: 23-33.

[16] Li Zhen, Jannesari A, Wolf F. An Efficient Data-Dependence Profiler for Sequential and Parallel Programs [C]//Proc of Parallel and Distributed Processing Symposium. Piscataway, NJ: IEEE Press, 2015: 484-493.